Table of Contents

1. Introduction
2. 前言
3. 实例变量、方法、类
4. 方法的调用
5. 实用元编程方法
6. 绑定
7. 块和绑定
8. 元编程实战
9. 结语

Ruby中的元编程

简介

本系列翻译自Ruby Metaprogramming站点上的课程笔记，并加入了我（DeathKing）的一些个人演绎、资料补充等。希望对大家有所帮助。

该课程由Satoshi Asakawa讲授，使用ruby 1.9.1p243 [i386-mingw32]。而我的测试环境则是ruby 1.9.2p180 [i386-mingw32]。

本系列教程在2011年9月间翻译完毕，最初发布在我的博客上。于2014年5月间，藉由GitBook整理，并发布在GitHub上。在此期间，Paolo Perrotta所著的《Ruby元编程》也被翻译并出版。同时，Ruby版本也从1.9飞跃到了2.1，尽管如此，本系列介绍的Ruby元编程技术仍然可用，读者可以放心阅读。

翻译的动机

Ruby是动态的、魔幻而有趣的。而元编程（Metaprogramming）技术在Ruby中的应用也让我大开眼界，虽然以前也有浅显地介绍反射机制（Reflection），但我仍然觉得才疏学浅，不能让大家完全感受到元编程的优美。借此机会，我想借Satoshi Asakawa的本系列讲座，为大家展示一个绚丽的Ruby元编程世界。

前言

元编程概览

元编程的定义看似是明确的，但却又模棱两可。维基百科上对元编程的定义如下：

> 元编程是指某类计算机程序的编写，这类计算机程序编写或者操纵其它程序（或者自身）作为它们的数据，或者在运行时完成部分本应在编译时完成的工作。多数情况下，与手工编写全部代码相比，程序员可以获得更高的工作效率, 或者给与程序更大的灵活度去处理新的情形而无需重新编译。

而我也在网上找到了Free Mind对元编程的简介：

> 回到元编程，程序处理程序可以分为“处理其他程序”和“处理自己”，对于前者，有我们熟悉的lex和yacc作为例子。而对于后者，如果再细分，可以分为“宏扩展”、“源代码生成”以及“运行时动态修改”等几种。
>
> 宏扩展从最简单的C语言的宏到复杂的Lisp的宏系统，甚至C++的“模板元编程”也可以包含在这一类里面，我在这里对它们进行了一些介绍。
>
> 源代码生成则主要是利用编程语言的eval功能，对生成出来的源代码（除了在Lisp这样的语言里面以外，通常是以字符串的方式）进行求值。有一类有趣的程序quine，它们运行的结果就是把自己的源代码原封不动地打印出来，通常要证明你精通某一门语言，为它写一个quine是绝佳的选择了，这里有我搜集的一些相关资料。
>
> 最后是运行时修改，像Lisp、Python以及Ruby之类的语言允许在运行时直接修改程序自身的结构，而不用再经过“生成源代码”这一层。当然对源代码进行eval的方法也许是最灵活的，但是它的效率却不怎么样，所以相来说并不是特别流行。这里主要介绍的是这种方式的元编程在Ruby里面的应用，如果对元编程的其他方面感兴趣，前面的几个链接应该都是很好的去处。

而元编程技术，最早来自于Lisp。John M. Vlissides在Pattern Languages of Program Design一书中写到：

> Lisp社团位于现在称之为“反射编程”（reflective programming）或“元编程”（metaprogramming）——对可编程语言编程的前沿。Smalltalk事实上从20世纪70年代后期就开始支持元类。但它是Lisp专用语言，从Comman和ObjVlisp开始，推动元编程成为主流[Bobrow+86，Cointe87]。早期工作为主对象协议（Metaobject Protocal）[Kiczales+91]打下了基础。主对象协议主要用来推广元编程。商业系统也开始使用元编程，特别是在IBM SOM平台上。

而你或许不知道Meta（源于希腊语）这个前缀在这里的意思是抽象。

Ruby中的元编程

Ruby中的元编程，是可以在运行时动态地操作语言结构（如类、模块、实例变量等）的技术。你甚至于可以在不用重启的情况下，在运行时直接键入一段新的Ruby代码，并执行它。

Ruby的元编程，也具有“利用代码来编写代码”的作用。例如，常见的 `attr_accessor` 等方法就是如此。

一点技术上的细节

> Lisp通过既可以用于代码也可以用于数据的S表达式（本质上是程序抽象语法树的直接翻译）支

> 持语法层的元编程。Lisp的元编程大量的使用了宏，宏的本质是代码模板。Lisp的这种方式带来的好处是可以在单一的层次上进行编程，代码及数据都以相同的方式表现，唯一的区别在于是否会被估值（evaluate）。然而语法层的元编程模式也有其弊端，用在同一命名空间下运行和估值的代码对两个抽象层次进行操作，会直接导致变量捕获（Variable Capture）和不经意的多次估值这类问题的出现。纵使有标准的Lisp惯用法可以处理这些问题，Lisp程序员仍然是需要学习和考虑更多的东西。
>
> Ruby可以通过ParseTree库来完成语法层的自省，ParseTree可以将Ruby源代码翻译成S表达式。使用此库来编写的一个有趣的应用叫做Heckle，它是一个“测试的测试”框架，能够对Ruby代码解析及改变，例如改变字符串或者将'true'和'false'进行来回的调换。其想法是如果测试覆盖率很好，那么对代码任何部分的任何变更都应该导致单元测试的失败。
>
> 与语法自省相对应的一种更高层次的自省叫做语义自省，既通过语言更高层次的数据结构对程序进行探查。在不同的编程语言中语义自省的方式十分不同，在Ruby语言中一般来说都是作用于类及方法层上：创建方法，重写方法，给方法赋予别名（alias）；截取方法调用；操纵继承链。这些技术和语法层自省相比与已有的代码更为正交（相关度更小），因为它们倾向于将已存在的方法视为黑盒而不是修改其内部实现。
>
> ——摘自里克的自习室

更多的参考资料

1. Ruby's Metaprogramming toolbox:http://weare.buildingsky.net/2009/08/25/rubys-metaprogramming-toolbox
2. Metaprogramming in Ruby:http://yehudakatz.com/2009/11/15/metaprogramming-in-ruby-its-all-about-the-self/
3. Practical Metaprogramming with Ruby: Storing Preferences:http://www.kalzumeus.com/2009/11/17/practical-metaprogramming-with-ruby-storing-preferences/
4. The Ruby Object Model and Metaprogramming:http://pragprog.com/screencasts/v-dtrubyom/the-ruby-object-model-and-metaprogramming
5. Metaprogramming Ruby: class_eval and instance_eval:http://jimmycuadra.com/posts/metaprogramming-ruby-class-eval-and-instance-eval
6. Metaprogramming in Ruby:http://rubyrogues.com/metaprogramming-in-ruby/

实例变量、方法、类

对象的实例变量及方法

实例变量（**Instance Variables**）是当你使用它们时，才会被建立的对象。因此，即使是同一个类的实例，也可以有不同的实例变量。

从技术层面上来看，一个对象（实例）只是存储了它的实例变量和其所属类的引用。因此，一个对象的实例变量仅存在于对象中，方法（我们称之为实例方法（**Instance Methods**））则存在于对象所属的类中。这也就是为什么同一个类的实例都共享类中的方法，却不能共享实例变量的原因了。

类

- 类也是对象。
- 因为类也是一个对象，能应用于对象的皆可运用于类。类和任何对象一样，有它们自己的类， `Class` 类即是 `Class` 类的实例。
- 与其它的对象一样， 类也有方法。对象的方法即是其所属类的实例方法。亦即，任何一个类的方法就是 `Class` 类的实例方法。
- 所有的类有共同的祖先 `Object` 类（都是从 `Object` 类直接或间接继承而来），而 `Object` 类又继承自 `BasicObject` 类，Ruby类的根本。
- 类名是常量（**Constant**）。

下面的代码有助于你理解这些信息：

```
# 对象的方法即为其所属类的实例方法
1.methods == 1.class.instance_methods
#=> true

# 类的“溯源”
N = Class.new
N.ancestors
#=> [N, Object, Kernel, BasicObject]
N.class
#=> Class
N.superclass
#=> Object
N.superclass.superclass
#=> BasicObject
N.superclass.superclass.superclass
#=> nil
```

类是开放的

在Ruby中， 类始终都是开放的。你可以重定义一个类，甚至于像 `String` 或 `Array` 这样的标准库中的类。（译注：Ruby 2.0引入了 `refine` 来限定这种打开类的作用域。）

```
class String
  def writesize
    self.size
  end
end

puts "Tell me my size!".writesize
```

> 注意!
>
> 　　当你打开一个类的时候，一定要万分小心！如果你随意向类中添加方法或数据，你可能会收到许多的BUG，比如，你定义了自己的`captalize()`方法并且漫不经心的覆盖了原`String`类中的`captalize()`方法，你很可能会收到风险。

多重initialize方法

　　下面将示范一个类的重载（**Overloading**）。我们编写了一个`Rectangule`类，该类用于将一个矩形呈现在网格上。当你在实例化一个`Rectangule`对象时，你可以使用两种方法：传递矩形的左上、右下的坐标，或者左上点坐标及矩形的宽度、高度。虽然Ruby中每个类只有一个`initialize`方法，但这种做法允许你的一个`initialize`就像两个不同的`initialize`一样。

```
# The Rectangle constructor accepts arguments in either
# of the following forms:
#   Rectangle.new([x_top, y_left], length, width)
#   Rectangle.new([x_top, y_left], [x_bottom, y_right])
class Rectangle
  def initialize(*args)
    if args.size < 2  || args.size > 3
      puts 'Sorry. This method takes either 2 or 3 arguments.'
    else
      puts 'Correct number of arguments.'
    end
  end
end
Rectangle.new([10, 23], 4, 10)
Rectangle.new([10, 23], [14, 13])
```

　　上述代码还不足以编写一个完整的`Rectangule`程序，但却足以演示方法重载是如何实现的。对`initialize`方法的重载可使得其具有处理可变参数的能力。这种技巧不但适用于`initialize`方法，也适用于其他方法！

匿名类

　　一个匿名类（**Anonymous Class**）也被称作单例类（**Singleton Class**），特征类（**Eigenclass**），鬼魂类（**Ghost Class**），元类（**Metaclass**）或者**uniclass**。

> 　　关于eigenclass这个名字的由来：大多数人叫它singlton classes，另一部分人叫它metaclasses，意思是the class of a class。但是这些名字都不足以描述它。Ruby之父Matz至今还没有给出一个官方的名字，但是他似乎喜欢叫它eigenclass，eigen这个词来自于德语，意思是one’s own。所以，eigenclass就被翻译为“an object’s own class”。而eigenclass的方法名字，则取了一个比较科学严肃的名字叫Singlton Methods。 ——参考自blackanger对Metaprogramming Ruby的笔记

> (5)](http://book.douban.com/people/blackanger/annotation/4086938/))
>
> “特征类”是一个很好的命名。这个“特征”和线性代数中“特征值”、“特征向量”的“特征”是一个意思。“特征值”、“特征向量”的名称是由德国数学家 David Hilbert （大卫·希尔伯特）在 1904 年使用并得以流传的，德语单词“Eigen”本意为“自己的”。Eigenclass，也就是意味着“自己的类”，用于 Ruby 的“单例类”概念之上十分贴切，因为“单例类”实际上就是一个对象独有的类。 ——参考自紫苏的特征类一文

Ruby中每个对象都有其自己的匿名类，一个类能拥有方法，但是只能对该对象本身其作用：当我们对一个具体的对象添加方法时，Ruby会插入一个新的匿名类于父类之间，来容纳这个新建立的方法。值得注意的是，匿名类通常是不可见（Hidden）的。它没有名字因此不能像其他类一样，通过一个常量来访问。你不能为这个匿名类实例化一个新的对象。

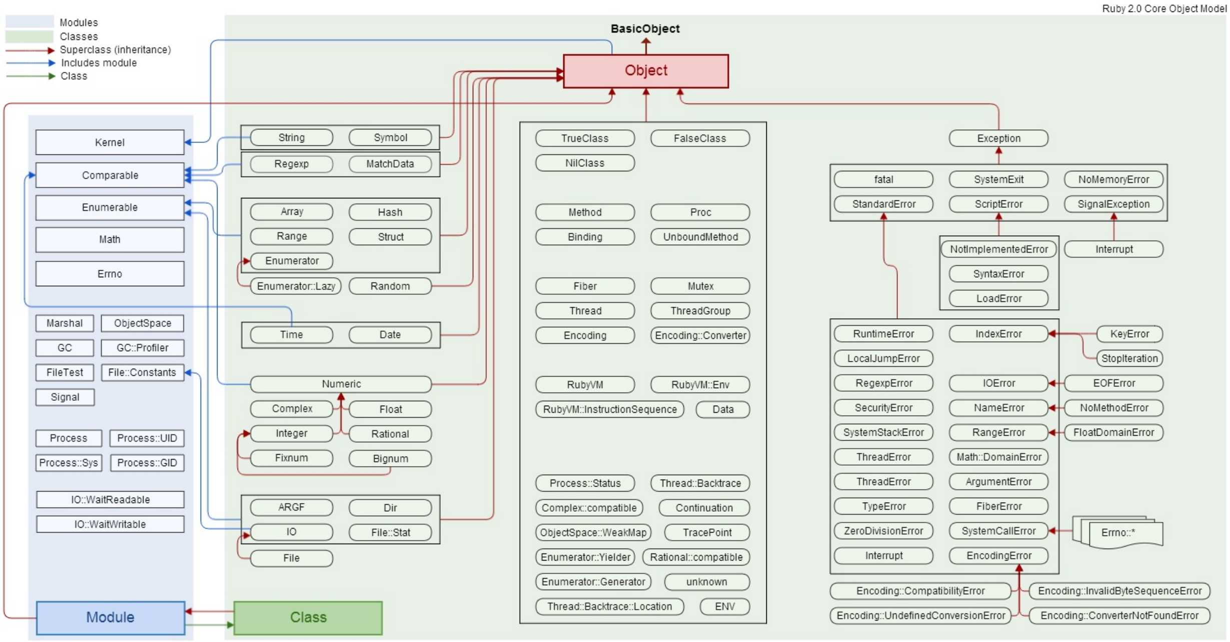
下面展示了建立匿名类的一些方法：

```
# 1
class Rubyist
  def self.who
    "Geek"
  end
end

# 2
class Rubyist
  class << self
    def who
      "Geek"
    end
  end
end

# 3
class Rubyist
end
def Rubyist.who
  "Geek"
end

#4
class Rubyist
end
Rubyist.instance_eval do
  def who
    "Geek"
  end
end
puts Rubyist.who # => Geek

# 5
class << Rubyist
  def who
    "Geek"
  end
end
```

上述5段代码，分别定义了`Rubylist.who`方法，该方法返回`"Geek"`。

任何时候，一旦你看到如上述标号为#5的代码， `class` 关键字后面紧接着 `<<` ，你就应该确信这里为 `<<` 右边的对象打开了一个匿名类。

你可以参看Complete Ruby Class Diagram 一文，文中展示了Ruby 1.8.6中的用户定义类及他们的超类的关系。而下面这幅由Artem S贡献的Ruby层次关系（Ruby 2.0.0）也非常有用：



方法的调用

当你在调用某一个方法的时候，Ruby会完成下面的步骤：

- 找到这个方法，我们把这个过程称作方法查找（**method lookup**）；
- 执行这个方法，为了执行这个方法，Ruby需要一个叫做`self`的伪变量；

方法的查找

要理解Ruby的方法查找，你需要了解下面两个概念：接受者（**receiver**）和祖先链（**ancestors chain**）。接受者就是方法的调用者。例如，对调用`an_object.display()`来说，`an_object`就是接受者。而关于祖先链，请考察任何一个Ruby类的内部。想象一下从一个类移动到其父类，然后再移动到父类的父类，直到到达`Object`类（默认的父类），然后继续移动，最终到达`BasicObject`类（Ruby层级模型中的根）。你遍历这些类所走过的路径就是"祖先链"（祖先链中也包括模块）。

因此，为了查找方法，Ruby首先进入receiver所属的类，并以此为起始，沿着祖先链不断前进，直到找到目标方法。这种行为也被称作"向右一步，再向上（**one step to the right, then up**）"规则：向右一步，进入reciver所属的类，然后再向上查找祖先链，直到找到目标方法。（译者注：如果遍历完祖先链也没有找到方法的话，会调用`method_missing`方法，如果这个方法没有被定义，则抛出`NoMethodError`）当你在一个类中包含一个模块时，Ruby创建了一个匿名类来封装这个模块，并将这个匿名类插入到祖先链中，也在这个类的上方。

下面的代码演示了一次方法查找：

```
class A
  def foo
  end
end
class B < A
  def bar
    # bar method in B
  end
end
class C < B
  def bar
    # bar method in C
    # overwriting superclass' method
  end
end

obj = C.new
obj.bar #=> in C class
obj.foo #=> not in C class
        #=>   then go to C's superclass B
        #=> not in B class
        #=>   then go to B's superclass A
        #=> execute it
```

self

- 在Ruby中，`self`是一个很特殊的变量，它总是指向当前对象（current object）；
- `self`被认为是默认的receiver。也就是说，你如果没有明确指出receiver，那么系统将把`self`当做

receiver；

- 实例变量是在`self`（当前对象）中查找的。也就是说，如果我使用`@var`，那么Ruby就会在`self`所指向的对象中去寻找此实例变量。需要注意的是，实例变量并不是由类定义的，它们也和子类以及继承机制无关。

  因此，当我们在调用一个明确指出receiver的方法时，Ruby会按照下面的步骤执行：

```
obj.do_method(param)
```

- 将`self`指向`obj`；
- 在`self`所属的类中查找`do_method(param)`方法（方法是存放在类中，而不是实例中！）；
- 调用方法`do_method(param)`；

实用元编程方法

本章节将介绍一系列的元编程实用方法，使读者对元编程有一个更为具体的认识。其中一些技术，诸如反射机制，已经有很多文章介绍过了，读者可以根据自身的情况进行选择是否跳过。

内省、反射

> 一说编写元程序的语言称之为元语言。被操纵的程序的语言称之为目标语言。一门编程语言同时也是自身的元语言的能力称之为反射或者自反。
>
> ——摘自维基百科元编程条目

在Ruby中，你完全有能力在运行时查看类或对象的信息。我们可以使用`class`、`instance_methods`、`intance_variables`等方法来达到目的。我们将这种技术称为内省（**Introspection**）或者反射（**Reflection**）。请考虑下面的代码：

```
class Rubyist
  def what_does_he_do
    @person = 'A Rubyist'
    'Ruby programming'
  end
end

an_object = Rubyist.new
puts an_object.class # => Rubyist
puts an_object.class.instance_methods(false) # => what_does_he_do
an_object.what_does_he_do
puts an_object.instance_variables # => @person
```

`respond_to?`方法是反射机制中另一个有用的方法。使用`respond_to?`方法，可以提前知道对象是否能够处理你想要交予它执行的信息。所有的对象都有此方法，使用`respond_to?`方法，你可以确定对象是否能使用指定的方法。

```
obj = Object.new
if obj.respond_to?(:program)
  obj.program
else
  puts "Sorry, the object doesn't understand the 'program' message."
end
```

send

`send`是`Object`类的实例方法。`send`方法的第一个参数是你期望对象执行的方法的名称。可以是一个字符串（**String**）或者符号（**Symbol**），但是我们更喜欢使用符号。剩余的参数就直接传递给所指定的方法。

```
class Rubyist
  def welcome(*args)
    "Welcome " + args.join(' ')
  end
end

obj = Rubyist.new
puts(obj.send(:welcome, "famous", "Rubyists"))   # => Welcome famous Rubyists
```

使用 `send` 方法，你所想要调用的方法就顺理成章的变成了一个普通的参数。你可以在运行时，直至最后一刻自由决定到底调用哪个方法。

```
class Rubyist
end

rubyist = Rubyist.new
if rubyist.respond_to?(:also_railist)
  puts rubyist.send(:also_railist)
else
  puts "No such information available"
end
```

上述代码展示了如果 `rubyist` 对象知道如何处理 `also_railist` 方法，那么他将会进行处理。

你可以通过 `send` 方法调用任何方法，即使是私有方法。

```
class Rubyist
  private
    def say_hello(name)
      "#{name} rocks!!"
    end
end

obj = Rubyist.new
puts obj.send(:say_hello, 'Matz')
```

define_method

Module#define_method是Module类实例的私有方法。因此define_method方法仅能由类或者模块使用。你可以通过define_method动态的在receiver中定义实例方法。而你仅需要传递需要定义的方法的名字，以及一个代码块（block），就如下面演示的那样：

```
class Rubyist
  define_method :hello do |my_arg|
    my_arg
  end
end

obj = Rubyist.new
puts(obj.hello('Matz')) # => Matz
```

method_missing

当Ruby使用look-up机制找寻方法时，如果方法不存在，那么Ruby将会在原receiver中自行调用一个叫做 `method_missing` 的方法。 `method_missing` 方法会以符号的形式传递被调用的那个不存在的方法的名字，以数组的形式传递调用时的参数，以及原调用中传递的块。 `method_missing` 是由 `Kernel` 模块提供的方法，因此任意对象都有此方法。 `Kernel#method_missing` 方法能响应 `NoMethodError` 错误。重载 `method_missing` 方法允许你对不存在的方法进行处理。

```
class Rubyist
  def method_missing(m, *args, &block)
    puts "Called #{m} with #{args.inspect} and #{block}"
  end
end

Rubyist.new.anything # => Called anything with [] and
Rubyist.new.anything(3, 4) { something } # => Called anything with [3, 4] and #<Proc:0x02efd664@tr
```

关于 `method_missing` ，ihower给出了一个漂亮的例子：

```
car = Car.new
car.go_to_taipei
# go to taipei
car.go_to_shanghai
# go to shanghai
car.go_to_japan
# go to japan

class Car
  def go(place)
    puts "go to #{place}"
  end

  def method_missing(name, *args)
    if name.to_s =~ /^go_to_(.*)/
      go($1)
    else
      super
    end
  end
end
```

这个例子出自于他在Ruby Conf China 2010上的讲义《如何设计出漂亮的Ruby API》中，你能够在他的个人博客中看到相关介绍，不过这个例子只出现在讲义的幻灯片中。

> 注意
>
> `method_missing` 方法的效率不甚理想，对效率敏感的项目尽量要避免使用此方法。尽管 `method_missing` 的确很强力。

remove_method和undef_method

想要移除已存在的方法，你可以在一个打开的类的作用域（**Scope**）内使用 `remove_method` 方法。即

使是父类以及父类的父类等先祖中有同名的方法，那些方法也不会被移除。而相比之下， `undef_method` 会阻止任何对指定方法的访问，无论该方法是在对象所属的类中被定义，还是其父类及其先祖类。

```
class Rubyist
  def method_missing(m, *args, &block)
    puts "Method Missing: Called #{m} with #{args.inspect} and #{block}"
  end

  def hello
    puts "Hello from class Rubyist"
  end
end

class IndianRubyist < Rubyist
  def hello
    puts "Hello from class IndianRubyist"
  end
end

obj = IndianRubyist.new
obj.hello # => Hello from class IndianRubyist

class IndianRubyist
  remove_method :hello  # removed from IndianRubyist, but still in Rubyist
end
obj.hello # => Hello from class Rubyist

class IndianRubyist
  undef_method :hello   # prevent any calls to 'hello'
end
obj.hello # => Method Missing: Called hello with [] and
```

eval

`Kernel` 模块提供了一个叫做 `eval` 的方法，该方法用于执行一个用字符串表示的代码。 `eval` 方法可以计算多行代码，使得将整个程序代码嵌入到字符串中并执行成为了可能。 `eval` 方法很慢，在执行字符串前最好对其预先求值。不过，糟糕的是， `eval` 方法会变得十分危险。如果外部数据通过 `eval` 传递的话，你就可能会遭遇一个安全漏洞，因为这些数据可能含有任意的代码，并且你的程序将会执行它。现在， `eval` 方法是在万般无奈的情况下才被选择的。

```
str = "Hello"
puts eval("str + ' Rubyist'") # => "Hello Rubyist"
```

关于 `eval` 方法，苏小脉给出了下面的建议：

> 一般来说，能避免 `eval` 就尽量避免，因为 `eval` 有额外的“分析时”开销（将字符串作为源代码进行词法、文法分析），而这个“剖析时”却又是在程序“运行时”进行的。把不需要惰性求值的表达式预先进行及早求值，能避免一些分析时开销。如果可能的话，用 `instance_exec` ，或 `instance_eval` 带块的形式，也比直接在字符串上求值好。
>
> ——苏小脉在如果用这种方式来构造一些复杂的对象呢？上的发言

而关于 `eval` 方法的安全性漏洞，Dave Thomas在他的著作Programming Ruby（英文页面）中列举了

一个十分有趣的例子：

> Walter Webcoder有一个非常棒的想法：设计一个Web算数页面。该页面是含有一个文本域以及按钮的简单Web表单，并被各种各样的非常酷的数学链接和横幅广告包围，使得看起来丰富多彩。用户输入一个算术表达式到文本域中，并按下按钮，然后结果就会被显示出来。一夜之间，世界上所有计算器都变得无用了；Walter大大获利，然后他退休并把他的余生用于收集车牌号。
>
> Walter认为实现这样一个计算器很容易。他可以用Ruby的 `CGI` 库访问表单域中的内容，再用 `eval` 方法把字符串当做表达式来求值。
>
> ```
> require 'cgi'
>
> cgi = CGI::new("html4")
>
> # Fetch the value of the form field "expression"
> expr = cgi["expression"].to_s
>
> begin
>  result = eval(expr)
> rescue Exception => detail
>  # handle bad expressions
> end
>
> # display result back to user...
> ```
>
> Walter把这个应用程序放到网上才几秒钟，来自Waxahachie的一个12岁小孩在表单中输入了 `system("rm")`，随他的计算机上的文件一起，Walter的美梦一下子破灭了。
>
> Walter得到了一个重要的教训：所有的外部数据都是有危险的。不要让它们靠近那些可能改动你的系统的接口。在这个案例中，表单中的内容是外部数据，而对 `eval` 的调用正是一个安全漏洞。
>
> ——Programming Ruby 中文第二版，Dave Thomas，Chad Fowler，Andy Hunt著

instance_eval, module_eval, class_eval

`instance_eval`， `module_eval` 和 `class_eval` 是 `eval` 方法的特殊形式。

instance_eval

`Object` 类提供了一个名为 `instance_eval` 的公开方法，该方法可被一个实例调用。他提供了操作对象的实例变量的途径。可以使用字符串向此方法传递参数或者传递一个代码块。

```ruby
class Rubyist
  def initialize
    @geek = "Matz"
  end
end
obj = Rubyist.new

# instance_eval可以操纵obj的私有方法以及实例变量

obj.instance_eval do
  puts self # => #<Rubyist:0x2ef83d0>
  puts @geek # => Matz
end
```

通过 `instance_eval` 传递的代码块使得你可以在对象内部操作。你可以在对象内部肆意操纵，不再会有任何数据是私有的！ `instance_eval` 亦可用于添加类方法。

```ruby
class Rubyist
end

Rubyist.instance_eval do
  def who
    "Geek"
  end
end

puts Rubyist.who # => Geek
```

还记得我们在之前匿名类的讲述（代码清单第四条）么？这个例子在这里被再一次的使用。

module_eval, class_eval

`module_eval` 和 `class_eval` 方法用于模块和类，而不是对象。 `class_eval` 是 `module_eval` 的一个别名。 `module_eval` 和 `class_eval` 可用于从外部检索类变量。

```ruby
class Rubyist
  @@geek = "Ruby's Matz"
end
puts Rubyist.class_eval("@@geek") # => Ruby's Matz
```

`module_eval` 和 `class_eval` 方法亦可用于添加类或模块的实例方法。尽管名字上两个方法时不同的，但他们的功能是相同的，并且都可以在模块或者类上使用。

```ruby
class Rubyist
end
Rubyist.class_eval do
  def who
    "Geek"
  end
end
obj = Rubyist.new
puts obj.who # => Geek
```

> 备注
>
> 当作用于类时，`class_eval`将会定义实例方法，而`instance_eval`定义类方法。

class_variable_get, class_variable_set

添加或查询一个类变量，`class_variable_get`方法和`class_variable_set`方法都可以被使用。`class_variable_get`方法需要一个代表变量名称的符号作为参数，并返回变量的值。`class_variable_set`方法也需要一个代表变量名称的符号作为参数，同时也要求传递一个值，作为欲设定的值。

```
class Rubyist
  @@geek = "Ruby's Matz"
end

Rubyist.class_variable_set(:@@geek, 'Matz rocks!')
puts Rubyist.class_variable_get(:@@geek) # => Matz rocks!
```

class_variables

如果你想知道一个类中有哪些类变量，我们可以使用`class_varibles方法`。他返回一个数组（**Array**），以符号（**Symbol**）的形式返回类变量的名称。

```
class Rubyist
  @@geek = "Ruby's Matz"
  @@country = "USA"
end

class Child < Rubyist
  @@city = "Nashville"
end
print Rubyist.class_variables # => [:@@geek, :@@country]
puts
p Child.class_variables # => [:@@city]
```

你可以从程序的输出中观察到`Child.class_variables`输出的是在`Child`类中定义的类变量（`@@city`）。`Child`类没有从父类中继承类变量（`@@geek`，`@@country`）。

instance_variable_get, instance_variable_set

我们可以使用`instance_variable_get`方法查询实例变量的值。

```
class Rubyist
  def initialize(p1, p2)
    @geek, @country = p1, p2
  end
end
obj = Rubyist.new('Matz', 'USA')
puts obj.instance_variable_get(:@geek) # => Matz
puts obj.instance_variable_get(:@country) # => USA
```

类比于 class_variable_set， 你可以使用 instance_variable_set 来设置一个对象的实例变量的值。

```
class Rubyist
  def initialize(p1, p2)
    @geek, @country = p1, p2
  end
end

obj = Rubyist.new('Matz', 'USA')
puts obj.instance_variable_get(:@geek) # => Matz
puts obj.instance_variable_get(:@country) # => USA
obj.instance_variable_set(:@country, 'Japan')
puts obj.inspect # => #<Rubyist:0x2ef8038 @country="Japan", @geek="Matz">
```

这样做的好处就是，你不需要使用 attr_accessor 等方法为访问实例变量建立接口。

const_get, const_set

类似的， const_get 和 const_set 用于操作常量。 const_get 返回指定常量的值：

```
puts Float.const_get(:MIN) # => 2.2250738585072e-308
```

const_set 为指定的常量设置指定的值，并返回该对象。如果常量不存在，那么他会创建该常量，就是下面示范的那样：

```
class Rubyist
end
puts Rubyist.const_set("PI", 22.0/7.0) # => 3.14285714285714
```

因为 const_get 返回常量的值，因此，你可以使用此方法获得一个类的名字并为这个类添加一个新的实例化对象的方法。这样使得我们有能力在运行时创建类并实例化其实例。

```
# Let us call our new class 'Rubyist'
# (we could have prompted the user for a class name)
class_name = "rubyist".capitalize
Object.const_set(class_name, Class.new)
# Let us create a method 'who'
# (we could have prompted the user for a method name)
class_name = Object.const_get(class_name)
puts class_name # => Rubyist
class_name.class_eval do
  define_method :who do |my_arg|
    my_arg
  end
end
obj = class_name.new
puts obj.who('Matz') # => Matz
```

绑定

诸如本地变量、实例变量、self一类的实体……或者说所有于对象绑定的名称。我们把他们称为绑定（**bindings**）。

下面内容摘自紫苏的博客，该文对我们的讨论很有意义。

在计算机科学中，“绑定”（**Binding**）一词是指一个更复杂、更大型的物件的引用的创建。例如当我们编写了一个函数，这个函数名就绑定了该函数本体，我们可以通过函数名来引用并调用该函数，这被称为名称绑定；又如当Ruby通过API去调用了C语言写的库函数时，这就是一个语言绑定；再如面向对象语言中的方法调度obj.method，这也是一个名称绑定，它会根据接收者obj具体的对象类型来确定应该引用哪个对象类型的method方法，而如果obj在编译时就能确定，那便可称之为静态绑定（早绑定），早期的静态类型语言（如C）使用的是早绑定；如果obj在运行时才能确定，那便可称为动态绑定（迟绑定），动态类型语言（如Ruby）使用的是迟绑定，而有些语言则同时支持早绑定和迟绑定，如C++的虚函数使用迟绑定，普通函数则使用早绑定。

在Ruby中，Kernel有一个方法binding，它返回一个Binding类型的对象。这个Binding对象就是我们这里说的绑定，它封装了当前执行上下文中的所有绑定（变量、方法、语句块、self的名称绑定），而这些绑定直接决定了面向对象语言中的执行环境。比如，当我们调用p时，实际上是进行了self和p的绑定，而p具体是哪个方法，是由self的类型来决定的，如果我们在顶层，而Kernel#p又没有被重写，那p就是一个用来显示对象细节的方法。可以说有了一个绑定的列表，我们就有了一个完整的面向对象上下文的拷贝，就好比上帝在12分37秒复制了一份世界，而这个世界与原本世界的环境一模一样，既有这朵花，又有那株草。Ruby的Binding对象的概念和 Continuation有共通之处，但Continuation主要用于实际堆、栈内存的环境跳转，而Binding则比较高层。

这个Binding对象有什么用？主要是用于eval这个函数。eval的第一个参数是需要eval的一段脚本字符串，而第二个可选参数则接受一个Binding对象。当指定了Binding时，eval会在传递给它的Binding所封装的执行环境里执行脚本，否则是在调用者的执行环境里执行。我们可以通过这个机制来进行一些不同上下文之间的通信，或者是在一个上下文即将被销毁之前保存该上下文环境以留他用，如：

```
def foo
  bar = 'baz'
  return binding
end

eval('p bar', foo)
```

这里我们通过foo返回的Binding获取到了局部上下文销毁前的局部变量bar的值，而在不使用binding的情况下，局部变量bar在foo外层是不可见的。

最后，Ruby有一个预定义的常量：TOPLEVEL_BINDING，它指向一个封装了顶层绑定的对象，通过它我们可以在其它上下文中通过eval在顶层上下文环境中执行脚本。

块和绑定

每一个Ruby块中，都包含了代码以及对应的绑定。当你定义一个块的时候，它会夺过当时环境的绑定，而当你执行一个带块的方法的时候，它又会传递这个绑定。

```
def who
  person = "Matz"
  yield("rocks")
end

person = "Matsumoto"

who do |y|
  puts("#{person}, #{y} the world") # => Matsumoto, rocks the world
  city = "Tokyo"
end

puts city
# => undefined local variable or method 'city' for main:Object (NameError)
```

观察上述代码，块附近的 `person` 变量在块被定义之前是一个新的局部变量，而不是 `who` 方法中的 `person` 变量。因此，块捕获了本地绑定并把它们结合在了一起。你可以在块中定义新的绑定，不过当块的生命周期结束后，它们也就消失了。

DeathKing注：注意变量定义的作用域。在块内部定义的变量不能用于块外部。而预先在块外部定义的变量，经过块的操作后，值会发生改变。

元编程实战

问题1

此问题改编自Dave Thomas的屏播Episode 5: Nine Examples of Metaprogramming。

众所周知，RubyLearnin.org的Core Ruby课程已经开办8周了。每周我们都有一个满分10分的测验。8周结束后，学生可以知道他的分数百分比。例如，有一个学生，在过去的8周里，他的得分情况为：5、10、10、10、10、10、10、10。那么，他的得分百分比为93.75%。

问题描述：每一批Core Ruby学习班有成百上千的学生。让我们假设我们有一个可以计算这个百分比并返回对应的值的Ruby方法。

现存的类和方法

让我们先来看看已存在的类和方法，并修改它，来解决上述问题。

```
class Result
  def total(*scores)
    percentage_calculation(*scores)
  end

  private

  def percentage_calculation(*scores)
    puts "Calculation for #{scores.inspect}"
    scores.inject {|sum, n| sum + n } * (100.0/80.0)
  end
end

r = Result.new
puts r.total(5,10,10,10,10,10,10,10)
puts r.total(5,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
```

上述代码中，我们定义了 `Result` 类以及一个 `total` 方法。 `total` 方法用于列举每个学生的成绩。 `score` 代表了学生在课程的8次竞赛中获得的成绩。私有方法 `percentage_calculation` 用于计算等分率。为了测试如此，我们调用 `total` 方法四次。前两次和后两次调用时分别采用相同的数组，运行程序，我们得到下面的输出：

```
Calculation for [5, 10, 10, 10, 10, 10, 10, 10]
93.75
Calculation for [5, 10, 10, 10, 10, 10, 10, 10]
93.75
Calculation for [10, 10, 10, 10, 10, 10, 10, 10]
100.0
Calculation for [10, 10, 10, 10, 10, 10, 10, 10]
100.0
```

观察输出，我们意识到我们调用了 `total` 方法四次，这也意味着我们调用 `percentage_calculation` 方法四次。我们将要尝试减少调用 `percentage_calculation` 方法的次数。

常规做法

减少对 percentage_calculation 方法的调用的一种途径是用某种方法存放之前计算出的数据。对此，我们需要定义一个叫做 MemoResult 的子类，该子类拥有一个叫做 @mem 的 Hash 类实例变量。代码如下：

```
class Result
  def total(*scores)
    percentage_calculation(*scores)
  end

  private

  def percentage_calculation(*scores)
    puts "Calculation for #{scores.inspect}"
    scores.inject {|sum, n| sum + n } * (100.0/80.0)
  end
end

class MemoResult < Result
  def initialize
    @mem = {}
  end
  def total(*scores)
    if @mem.has_key?(scores)
      @mem[scores]
    else
      @mem[scores] = super
    end
  end
end

r = MemoResult.new
puts r.total(5,10,10,10,10,10,10,10)
puts r.total(5,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
```

Hash 类提供了 has_key? 方法用于检查某个散列是否拥有指定的键。在上述程序中，如果 has_key? 返回 true，那么我们就直接使用 @mem 中存放的值，否则我们将调用 percentage_calculation(*scores) 重新计算改值并存放在 @mem 中。输出如下：

```
Calculation for [5, 10, 10, 10, 10, 10, 10, 10]
93.75
93.75
Calculation for [10, 10, 10, 10, 10, 10, 10, 10]
100.0
100.0
```

观察上述输出，我们不难得出，在第二次、第四次调用 r.total 的时候，我们直接使用的存储值。

使用**Class.new**和**define_method**的做法

上面使用的 MemoResult 类与其父类精密相连。为了避免这样，我们使用目前学过的Ruby元编程知识动态创建这个子类。

我们需要编写一个接受两个参数的方法 mem_result ：一个参数为父类的名字，另一个参数为方法的名字（而 mem_result 方法会返回定义好的类的名字）。下面是代码：

```
class Result
  def total(*scores)
    percentage_calculation(*scores)
  end

  private

  def percentage_calculation(*scores)
    puts "Calculation for #{scores.inspect}"
    scores.inject {|sum, n| sum + n } * (100.0/80.0)
  end
end

def mem_result(klass, method)
  mem = {}
  Class.new(klass) do
    define_method(method) do |*args|
      if mem.has_key?(args)
        mem[args]
      else
        mem[args] = super
      end
    end
  end
end

r = mem_result(Result, :total).new
puts r.total(5,10,10,10,10,10,10,10)
puts r.total(5,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
```

输出如下：

```
Calculation for [5, 10, 10, 10, 10, 10, 10, 10]
93.75
93.75
Calculation for [10, 10, 10, 10, 10, 10, 10, 10]
100.0
100.0
```

代码 Class.new(klass) 以给定的 klass 为父类，创建了一个匿名类。块被用作类的体，包含了该类中的方法。而 define_method 定义了 method 所代表的方法（也就是 mem_result 的第二个参数）。

> 注意
>
> 我们并没有编写 initialize 方法和使用实例变量 @mem 。相反地，我们使用的是局部变量 mem ，这是因为块是一个闭包，而局部变量 mem 在块内部是有效的。

使用匿名类的做法

```ruby
class Result
  def total(*scores)
    percentage_calculation(*scores)
  end

  private

  def percentage_calculation(*scores)
    puts "Calculation for #{scores.inspect}"
    scores.inject {|sum, n| sum + n } * (100.0/80.0)
  end
end

r = Result.new

# Anonymous class on object
def r.total(*scores)
  @mem ||= {}
  if @mem.has_key?(scores)
    @mem[scores]
  else
    @mem[scores] = super
  end
end

puts r.total(5,10,10,10,10,10,10,10)
puts r.total(5,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
```

使用即时创建匿名类的做法

```
class Result
  def total(*scores)
    percentage_calculation(*scores)
  end

  private

  def percentage_calculation(*scores)
    puts "Calculation for #{scores.inspect}"
    scores.inject {|sum, n| sum + n } * (100.0/80.0)
  end
end

def mem_result(obj, method)
  obj.class.class_eval do
    mem ||= {}
    define_method(method) do |*args|
      if mem.has_key?(args)
        mem[args]
      else
        mem[args] = super
      end
    end
  end
end

r = Result.new
mem_result(r, :total)

puts r.total(5,10,10,10,10,10,10,10)
puts r.total(5,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
```

上述的代码中，我们编写了新的`mem_result`方法。该方法的第一个参数`obj`接收一个对象，用于建立此对象的匿名类。该方法的第二个参数`method`接受一个方法，用于指明要在匿名类中创建的方法名字。

在此之前，我们已经使用`define_method`即时地创建了一个方法。问题是，`define_method`只能对类或模块起作用，而我们这里处理的是对象。因此，我们使用`obj.class`来获得对象所属的类，然后对类使用`class_eval`和`define_method`方法在该类中添加一个实例方法`method`。现在，我们来运行一下代码并检查输出。

```
result.rb:21:in `total': super: no superclass method `total' (NoMethodError)
  from result.rb:30
```

代码没有运行。

代码`mem[args] = super`尝试在匿名类中调用`Result`类中的`total`方法。但问题是，我们是在`Result`类中直接定义的`total`方法。我们说过，`obj.class`返回的是`Result`，所以这并不起效。我们需要做得是创建一个匿名类，并将这个`total`方法放到这个匿名类中。同时，我们的匿名类应该是`Result`类的子类。

让我们像下面一样创建需要的匿名类。

```
anon = class << obj
  self
end
```

上述代码中的`self`返回了我们需要的匿名类，并被变量`anon`所引用。大多数的Ruby程序员会像下面这样把这些代码写作一行，以表示他们正创建鬼魂类。

```
anon = class << obj; self; end
```

有了匿名类后，我们应该对其使用`class_eval`方法，就像下面这样：

```
class Result
  def total(*scores)
    percentage_calculation(*scores)
  end

  private

  def percentage_calculation(*scores)
    puts "Calculation for #{scores.inspect}"
    scores.inject {|sum, n| sum + n } * (100.0/80.0)
  end
end

def mem_result(obj, method)
  anon = class << obj; self; end
  anon.class_eval do
    mem ||= {}
    define_method(method) do |*args|
      if mem.has_key?(args)
        mem[args]
      else
        mem[args] = super
      end
    end
  end
end

r = Result.new
mem_result(r, :total)

puts r.total(5,10,10,10,10,10,10,10)
puts r.total(5,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
puts r.total(10,10,10,10,10,10,10,10)
```

代码正常运行，并返回希望的结果。

问题2

本例根据Hal Fulton的文章[An Exercise in Metaprogramming with Ruby(http://an%20exercise%20in%20metaprogramming%20with%20ruby/)改编。

假设我们有两个CSV（comma-separated values，数据使用逗号分隔的）文件，其头部有一些描述性

的文字，如下所示：

文件：location.txt

```
name,country
"Matz", "USA"
"Fabio Akita", "Brazil"
"Peter Cooper", "UK"
```

文件：twitter.txt

```
twitterid,url
"AkitaOnRails","http://www.akitaonrails.com/"
"peterc","http://www.petercooper.co.uk/"
```

首先，我们建立一个名为datawrapper.rb的文件，并建立一个类。我们会调用 `DataWrapper` 类并定义一个名为 `wrap` 的类方法，此方法请求一个用于标识文件名的参数，并据此建立一个类。上面的两个文件的第一行都是由逗号分隔的属性名（attribute names）。因此，我们想要把这些文件当做是数据数组，我们将要读取这些数据，并以数组的形式存放。

```
# file: datawrapper.rb
class DataWrapper
   def self.wrap(file_name)
     data = File.new(file_name)
     header = data.gets.chomp
     data.close
     puts header # => name,country
     # in the end we return the class name
  end
end
```

接着，我们编写一个小程序名为testdatawrapper.rb。尝试着读取location.txt文件。

```
#testdatawrapper.rb
require 'datawrapper'
DataWrapper.wrap("location.txt")
```

回到datawrapper.rb程序，我们需要新建一个类，并给他适当的名字：

```
# file: datawrapper.rb
class DataWrapper
  def self.wrap(file_name)
    data = File.new(file_name)
    header = data.gets.chomp
    data.close
    class_name = File.basename(file_name, ".txt").capitalize
    klass = Object.const_set(class_name, Class.new)
    klass # we return the class name
  end
end
```

变量klass指代的是我们新建的类。如果file_name参数所指向的文件为“location.txt”，那么新建立得类名会被命名为Location。

再次运行修改后的程序。

```
#testdatawrapper.rb
require 'datawrapper'
data = DataWrapper.wrap("location.txt") # Capture return value and
puts data # => Location
```

然后就是大展身手的时候了。文件第一行读入的是表名（List Name）。我们只需要使用split方法即可快速读入。 修改后的datawrapper.rb如下：

```
# file: datawrapper.rb
class DataWrapper
  def self.wrap(file_name)
    data = File.new(file_name)
    header = data.gets.chomp
    data.close
    class_name = File.basename(file_name, ".txt").capitalize
    klass = Object.const_set(class_name, Class.new)
    # get attribute names
    names = header.split(",")
    p names # => ["name", "country"]
    klass # we return the class name
  end
end
```

现在，在我们新建立的类的上下文中使用class_eval。此时，我们会定义一个initialize方法。并且，我们应该建立一个to_s方法，使得我们能够将其输出，记得使用alias关键字将to_s方法作为inspect方法的同义词。修改后的datawrapper.rb程序如下：

```ruby
# file: datawrapper.rb
class DataWrapper
  def self.wrap(file_name)
    data = File.new(file_name)
    header = data.gets.chomp
    data.close
    class_name = File.basename(file_name, ".txt").capitalize
    klass = Object.const_set(class_name, Class.new)
    # get attribute names
    names = header.split(",")
    klass.class_eval do
      attr_accessor *names
      define_method(:initialize) do |*values|
        names.each_with_index do |name, i|
          instance_variable_set("@"+name, values[i])
        end
      end
      define_method(:to_s) do
        str = "<#{self.class}:"
        names.each {|name| str << " #{name}=#{self.send(name)}" }
        str + ">"
      end
      alias_method :inspect, :to_s
    end
    klass # we return the class name
  end
end
```

下一步，建立一个类方法，用于读取整个文本，并返回一个代表文本中内容的素组对象。类方法的建立涉及到了单体类等概念，但此处不需要仔细考察。修改后的代码如下：

```
# file: datawrapper.rb
class DataWrapper
  def self.wrap(file_name)
    data = File.new(file_name)
    header = data.gets.chomp
    data.close
    class_name = File.basename(file_name, ".txt").capitalize
    klass = Object.const_set(class_name, Class.new)
    # get attribute names
    names = header.split(",")
    klass.class_eval do
      attr_accessor *names
      define_method(:initialize) do |*values|
        names.each_with_index do |name, i|
          instance_variable_set("@"+name, values[i])
        end
      end
      define_method(:to_s) do
        str = "<#{self.class}:"
        names.each {|name| str << " #{name}=#{self.send(name)}" }
        str + ">"
      end
      alias_method :inspect, :to_s
    end
    def klass.read
      array = []
      data = File.new(self.to_s.downcase+".txt")
      data.gets  # throw away header
      data.each do |line|
        line.chomp!
        values = eval("[#{line}]")
        array << self.new(*values)
      end
      data.close
      array
    end
    klass # we return the class name
  end
end
```

修改testdatawrapper.rb并测试。

```
#testdatawrapper.rb
require 'datawrapper'
klass = DataWrapper.wrap("location.txt")
list = klass.read
list.each do |location|
  puts("#{location.name} is from the #{location.country}")
end
```

让我们来看看和location.txt文件完全不同的twitter.txt文件。针对于twitter.txt的测试文件如下：

```
#testdatawrapper.rb
require 'datawrapper'
klass = DataWrapper.wrap("twitter.txt")
list = klass.read
list.each do |twitter|
  puts("#{twitter.twitterid}'s site is #{twitter.url}")
end
```

即使我们使用了不同的数据，datawrapper.rb的代码也无需改变！这便是一个Ruby之中的元编程的例子与实践。

结语

Ruby元编程应用远不止如此，Rails里就大量涉及了这种技术。你可以仔细研读Rails的代码。

参考资料

- An Exercise in Metaprogramming with Ruby.
- Metaprogramming Ruby - Author: Paolo Perrotta.
- Metaprogramming in Ruby: It’s All About the Self.
- Programming Ruby 1.9 - Author: Dave Thomas.
- Seeing Metaclasses Clearly.
- The Book Of Ruby - Author: Huw Collingbourne.
- The Ruby Object Model and Metaprogramming screencasts with Dave Thomas.
- Understanding Ruby Singleton Classes.

其他参考

- 倾国剑气
- ihower
- Free Mind
- 里克的自习室